**Question 1**

**Given the following function check()which is passed an integer array, arrA[], and its associated size, size, and returns a Boolean (line numbers are included):**

L1. bool check(int arrA[], int size)

L2. {

L3. bool checked = true;

L4. for (int i = 0; i < size - 1 && checked; i++)

L5. {

L6. if (arrA[i] > arrA[i + 1])

L7. {

L8. checked = false;

L9. }

L10. }

L11. return (checked);

L12. }

1. **Explain, in your own words, and with reference to the appropriate line numbers, and some sample data, what the function check() does and how it works**

Function check takes 2 parameters: array of numbers and size of this array.

In **L3** Boolean variable **checked** is initialized with value true. It is used to determine if array is in ascending order. It is also used in **L4** inside the **for loop** as an condition to stop the loop in case the numbers are not in ascending order.

In **L4** with have a **for loop** that is iterating through the array from index 0 to size -1 to not exceed the array size, but when the variable checked is equals **false** the for loop will stop iterating.

In **L6** there **is if statement** that checks if the values in the array are not ascending order then it changes value of **checked** to **false** which results in stopping the for loop.

Then in **L11** the function returns the value of variable **checked**. If **checked** **= true** then the array is sorted in ascending order, if **checked = false** then the array is not sorted in ascending order.

**Suppose we have an array arrA[] = {3, 6, 8, 10, 12}.**

**Initialization (Line 3):**

checked is initialized to true.

**Looping through the Array (Lines 4-10):**

Iteration 1 (i = 0):

Check if arrA[0] (3) is greater than arrA[1] (6). It's not, so continue.

Iteration 2 (i = 1):

Check if arrA[1] (6) is greater than arrA[2] (8). It's not, so continue.

Iteration 3 (i = 2):

Check if arrA[2] (8) is greater than arrA[3] (10). It's not, so continue.

Iteration 4 (i = 3):

Check if arrA[3] (10) is greater than arrA[4] (12). It's not, so continue.

Loop ends because i = size - 1, and checked remains true.

**Return Statement (Line 11):**

**The function returns true, indicating that the array is sorted in ascending order.**

**So, in this example, the function correctly determines that the array {3, 6, 8, 10, 12} is sorted in ascending order.**

1. **Perform a time step analysis of the function check() as a function of the size of arrA[], in a best case and a worst case situation. Clearly explain any assumptions made.**

**Worst case** scenario, array is sorted in ascending order, so we have to each element

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Line | Cost | numTimes | cost\*numTimes | Total |
| 3 | 1 | 1 | 1 |  |
| 4 | 1 | N | N |  |
| 6 | 1 | N | N |  |
| 8 | 1 | 1 | 1 |  |
| 11 | 1 | 1 | 1 |  |
|  |  |  |  | f(N) = 2N+3 |

In a **best case** scenario first two elements already wont be in ascending order so the bool checked =false which will cause the for loop to stop.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Line | Cost | numTimes | cost\*numTimes | Total |
| 3 | 1 | 1 | 1 |  |
| 4 | 1 | 2 | 2 |  |
| 6 | 1 | 1 | 1 |  |
| 8 | 1 | 1 | 1 |  |
| 11 | 1 | 1 | 1 |  |
|  |  |  |  | f(N) = 5 |

**Question 2**

**Given the following function search() which is passed the integer arrays, arrA[], with non-unique, unsorted values of size, size, and an empty integer array, arrOccur[], the same size as arrA[]. Also passed to the function is an integer value item. (Note line numbers are also included):**

L1. int search(int arrA[], int size, int item, int arrOccur[])

L2. {

L3. int location = -1;

L4. for (int i = 0; i < size; i++)

L5. {

L6. if (arrA[i] == item)

L7. {

L8. ++location;

L9. arrOccur[location] = i;

L10. }

L11. }

L12. return (location);

L13. }

Function search is a function that searches for an item in non-unique array. It stores index of wanted item in array with occurrences and returns number of occurrences.

In L3 there is initialized variable location that has value -1, but if the item occures in array it will return occurrence count.

In L4 there is an for loop that goes through the array, then in L6 if statemnt checks if elemnt from the array is equal to an item to search.

L8 is responsible for couting occurences and L9 saves an index of element from an array that is equal to the item it serches for.

L12 return the number of occurrences.

**Suppose we have an array arrA[] = {22, 25, 26, 24, 25, 23, 25}**

**Initialization (Line 3):**

location is initialized to -1.

**Looping through the Array (Lines 4-11):**

Iteration 1 (i = 0):

Check if arrA[0] (22) is equal to item (25). It's not, so continue.

Iteration 2 (i = 1):

Check if arrA[1] (25) is equal to item (25). It is, so:

Increment location to 0.

Store index 1 in arrOccur[].

Iteration 3 (i = 2):

Check if arrA[2] (26) is equal to item (25). It's not, so continue.

Iteration 4 (i = 3):

Check if arrA[3] (24) is equal to item (25). It's not, so continue.

Iteration 5 (i = 4):

Check if arrA[4] (25) is equal to item (25). It is, so:

Increment location to 1.

Store index 4 in arrOccur[].

Iteration 6 (i = 5):

Check if arrA[5] (23) is equal to item (25). It's not, so continue.

Iteration 7 (i = 6):

Check if arrA[6] (25) is equal to item (25). It is, so:

Increment location to 2.

Store index 6 in arrOccur[].

**Return Statement (Line 12):**

The function returns location, which is 3 (the number of occurrences found).

**So, in this example, the function correctly determines that the temperature 25 occurs 3 times in the array {22, 25, 26, 24, 25, 23, 25}.**

**ii) Perform a time step analysis of the function search() as a function of the size of arrA[] in a worst case situation. Clearly explain any assumptions made.**

In the **worst-case** scenario, is when all elements in the array are the same and are equal to the item searched for.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Line | Cost | numTimes | cost\*numTimes | Total |
| 3 | 1 | 1 | 1 |  |
| 4 | 1 | N | N |  |
| 6 | 1 | N | N |  |
| 8 | 1 | N | N |  |
| 9 | 1 | N | N |  |
| 13 | 1 | 1 | 1 |  |
|  |  |  |  | f(N)=4N+2 |

**Question 3**

**Given two test files, file1.txt and file2.txt, (both with 10,000 integers), perform a comparison of the four sorting techniques considered for different values of N (array size) in terms of:**

**a. time taken**

**b. number of swaps/data moves**

**c. number of comparisons**

**Present your results in a meaningful and clear way so that it is easy to see differences between the algorithms for the same value of N.**

In my testing approach I was evaluating 4 sorting algorithms. Testing is performed on array of different size ranging from 1000 to 10000 in increments of 1000, for each size. Numbers are read from given files, first from file1, then from file2.

Counting swaps and comparisons is executed in each soring algorithm. The time taken to sort the array is measured by recording the time before and after the sorting algorithm is executed, and then calculating the difference.

After sorting an array and gathering all data, the sorted array is checked to ensure that it is sorted correctly.

Result are being save to .csv file.

Below I’ve included analysis of gathered data and parts of code. Full code and csv files are available in the gist at this link: <https://gist.github.com/Olszewski-Jakub/de67d4da5595c7e1c763ecce010af7b0>

**File 1**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Size** | **Bubble Sort** | **Count Sort** | **Insertion Sort** | **Selection Sort** |
| 1000 | 0,003 | 0 | 0,001 | 0,002 |
| 2000 | 0,012 | 0 | 0,004 | 0,007 |
| 3000 | 0,03 | 0 | 0,009 | 0,014 |
| 4000 | 0,06 | 0 | 0,015 | 0,025 |
| 5000 | 0,1 | 0 | 0,024 | 0,039 |
| 6000 | 0,148 | 0 | 0,036 | 0,061 |
| 7000 | 0,213 | 0 | 0,048 | 0,077 |
| 8000 | 0,275 | 0,001 | 0,077 | 0,098 |
| 9000 | 0,349 | 0 | 0,08 | 0,125 |
| 10000 | 0,44 | 0 | 0,109 | 0,15 |

**File 2**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Size** | **Bubble Sort** | **Count Sort** | **Insertion Sort** | **Selection Sort** |
| **1000** | 0,005 | 0 | 0,001 | 0,001 |
| 2000 | 0,016 | 0 | 0,003 | 0,006 |
| 3000 | 0,041 | 0 | 0,007 | 0,014 |
| 4000 | 0,072 | 0 | 0,014 | 0,024 |
| 5000 | 0,107 | 0 | 0,022 | 0,038 |
| 6000 | 0,189 | 0,001 | 0,031 | 0,052 |
| 7000 | 0,215 | 0 | 0,04 | 0,07 |
| 8000 | 0,294 | 0 | 0,053 | 0,09 |
| 9000 | 0,425 | 0 | 0,067 | 0,111 |
| 10000 | 0,405 | 0 | 0,082 | 0,132 |

**Data obtained from the code**

**File 1**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sorting Algorithm** | **Size** | **Swaps** | **Comparisons** | **Time Taken** |
| Bubble Sort | 1000 | 254554 | 499500 | 0.003000 |
| Bubble Sort | 2000 | 1001456 | 1999000 | 0.012000 |
| Bubble Sort | 3000 | 2237298 | 4498500 | 0.030000 |
| Bubble Sort | 4000 | 3934058 | 7998000 | 0.060000 |
| Bubble Sort | 5000 | 6138975 | 12497500 | 0.100000 |
| Bubble Sort | 6000 | 8833508 | 17997000 | 0.148000 |
| Bubble Sort | 7000 | 11949020 | 24496500 | 0.213000 |
| Bubble Sort | 8000 | 15847752 | 31996000 | 0.275000 |
| Bubble Sort | 9000 | 20218657 | 40495500 | 0.349000 |
| Bubble Sort | 10000 | 25154469 | 49995000 | 0.440000 |
| Selection Sort | 1000 | 999 | 499500 | 0.002000 |
| Selection Sort | 2000 | 1999 | 1999000 | 0.007000 |
| Selection Sort | 3000 | 2999 | 4498500 | 0.014000 |
| Selection Sort | 4000 | 3999 | 7998000 | 0.025000 |
| Selection Sort | 5000 | 4999 | 12497500 | 0.039000 |
| Selection Sort | 6000 | 5999 | 17997000 | 0.061000 |
| Selection Sort | 7000 | 6999 | 24496500 | 0.077000 |
| Selection Sort | 8000 | 7999 | 31996000 | 0.098000 |
| Selection Sort | 9000 | 8999 | 40495500 | 0.125000 |
| Selection Sort | 10000 | 9999 | 49995000 | 0.150000 |
| Insertion Sort | 1000 | 255552 | 254554 | 0.001000 |
| Insertion Sort | 2000 | 1003454 | 1001456 | 0.004000 |
| Insertion Sort | 3000 | 2240294 | 2237298 | 0.009000 |
| Insertion Sort | 4000 | 3938052 | 3934058 | 0.015000 |
| Insertion Sort | 5000 | 6143969 | 6138975 | 0.024000 |
| Insertion Sort | 6000 | 8839502 | 8833508 | 0.036000 |
| Insertion Sort | 7000 | 11956014 | 11949020 | 0.048000 |
| Insertion Sort | 8000 | 15855746 | 15847752 | 0.077000 |
| Insertion Sort | 9000 | 20227651 | 20218657 | 0.080000 |
| Insertion Sort | 10000 | 25164463 | 25154469 | 0.109000 |
| Count Sort | 1000 | 0 | 0 | 0.000000 |
| Count Sort | 2000 | 0 | 0 | 0.000000 |
| Count Sort | 3000 | 0 | 0 | 0.000000 |
| Count Sort | 4000 | 0 | 0 | 0.000000 |
| Count Sort | 5000 | 0 | 0 | 0.000000 |
| Count Sort | 6000 | 0 | 0 | 0.000000 |
| Count Sort | 7000 | 0 | 0 | 0.000000 |
| Count Sort | 8000 | 0 | 0 | 0.001000 |
| Count Sort | 9000 | 0 | 0 | 0.000000 |
| Count Sort | 10000 | 0 | 0 | 0.000000 |

**File 2**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sorting Algorithm** | **Size** | **Swaps** | **Comparisons** | **Time Taken** |
| Bubble Sort | 1000 | 250102 | 499500 | 0.005000 |
| Bubble Sort | 2000 | 995574 | 1999000 | 0.016000 |
| Bubble Sort | 3000 | 2227207 | 4498500 | 0.041000 |
| Bubble Sort | 4000 | 3977576 | 7998000 | 0.072000 |
| Bubble Sort | 5000 | 6187299 | 12497500 | 0.107000 |
| Bubble Sort | 6000 | 8938260 | 17997000 | 0.189000 |
| Bubble Sort | 7000 | 12177890 | 24496500 | 0.215000 |
| Bubble Sort | 8000 | 15914238 | 31996000 | 0.294000 |
| Bubble Sort | 9000 | 20158364 | 40495500 | 0.425000 |
| Bubble Sort | 10000 | 24712201 | 49995000 | 0.405000 |
| Selection Sort | 1000 | 999 | 499500 | 0.001000 |
| Selection Sort | 2000 | 1999 | 1999000 | 0.006000 |
| Selection Sort | 3000 | 2999 | 4498500 | 0.014000 |
| Selection Sort | 4000 | 3999 | 7998000 | 0.024000 |
| Selection Sort | 5000 | 4999 | 12497500 | 0.038000 |
| Selection Sort | 6000 | 5999 | 17997000 | 0.052000 |
| Selection Sort | 7000 | 6999 | 24496500 | 0.070000 |
| Selection Sort | 8000 | 7999 | 31996000 | 0.090000 |
| Selection Sort | 9000 | 8999 | 40495500 | 0.111000 |
| Selection Sort | 10000 | 9999 | 49995000 | 0.132000 |
| Insertion Sort | 1000 | 251091 | 250102 | 0.001000 |
| Insertion Sort | 2000 | 997557 | 995574 | 0.003000 |
| Insertion Sort | 3000 | 2230186 | 2227207 | 0.007000 |
| Insertion Sort | 4000 | 3981548 | 3977576 | 0.014000 |
| Insertion Sort | 5000 | 6192266 | 6187299 | 0.022000 |
| Insertion Sort | 6000 | 8944220 | 8938260 | 0.031000 |
| Insertion Sort | 7000 | 12184844 | 12177890 | 0.040000 |
| Insertion Sort | 8000 | 15922188 | 15914238 | 0.053000 |
| Insertion Sort | 9000 | 20167305 | 20158364 | 0.067000 |
| Insertion Sort | 10000 | 24722134 | 24712201 | 0.082000 |
| Count Sort | 1000 | 0 | 0 | 0.000000 |
| Count Sort | 2000 | 0 | 0 | 0.000000 |
| Count Sort | 3000 | 0 | 0 | 0.000000 |
| Count Sort | 4000 | 0 | 0 | 0.000000 |
| Count Sort | 5000 | 0 | 0 | 0.000000 |
| Count Sort | 6000 | 0 | 0 | 0.001000 |
| Count Sort | 7000 | 0 | 0 | 0.000000 |
| Count Sort | 8000 | 0 | 0 | 0.000000 |
| Count Sort | 9000 | 0 | 0 | 0.000000 |
| Count Sort | 10000 | 0 | 0 | 0.000000 |

**Sorting algorithms**

**A screenshot of a computer program

Description automatically generated**

**A screen shot of a computer program

Description automatically generated**

**A screenshot of a computer program

Description automatically generated**

**A screenshot of a computer program

Description automatically generated**

**A screenshot of a computer program

Description automatically generated**

**A computer screen shot of a program

Description automatically generated**

**I am aware of what plagiarism is and include this here to confirm that this work is my own**

